Locating Floating-Point Exceptions on the SGI Origin 2000

Mark R. Fahey*
U.S. Army Engineer Research and Development Center (ERDC)
Major Shared Resource Center (MSRC)
Vicksburg, MS

February 26, 2001

1 Introduction

The Institute of Electrical and Electronics Engineers (IEEE) standard 754, published in 1985 [7], defines a binary floating point arithmetic system. It is the product of severval years' effort by a working group of a subcommittee of the IEEE Computer Society Computer Standards Committee.

Every IEEE arithmetic operation produces a result whether it is mathematically correct or not. When an exceptional condition like division by zero or overflow occurs in IEEE arithmetic, the default response is to deliver a result and continue processing.

An exception is not an error unless handled poorly. To quote Kahan [8]:

> "Exceptions that reveal errors are merely messengers. What turns an exception into an error is bad exception-handling."

What makes exceptional operations exceptional is that no single default response can be satisfactory in every instance. On the other hand, if a default response will serve most instances satisfactorily, the unsatisfactory instances cannot justify aborting computation every time the exception occurs. The IEEE arithmetic system is designed to continue to function on a computation as long as possible, handling unusual situations with reasonable default responses, including setting appropriate flags.

For many programmers, a floating-point exception is a sign of a "buggy" program. These "bugs" can range from use of uninitialized variables to incorrectly coded algorithms. These same programmers often want to determine exactly where unexpected exceptions occur and "fix" the code to avoid these occurrences. The purpose of this paper is to demonstrate how to detect and locate arithmetic exceptions on the SGI Origin 2000 (O2K). The O2K has several features to handle floating-point exceptions, and each of them will be discussed. This report additionally serves as an introduction to exception-handling techniques.

In Section 2, the IEEE Standard for Binary Floating-Point Arithmetic is reviewed as well as the floating-point representation on the O2K. In Section 3, floating-point exceptions are discussed, and methods for detecting them are presented. In Section 4, several methods for locating exceptions with various programming tools are demonstrated.

*Computational Migration Group, Computer Sciences Corporation, mfahey@wes.hpc.mil

2 IEEE Floating-Point Review

The IEEE Standard for binary arithmetic [7] defines the most commonly used floating-point system. The standard specifies floating-point number formats, the results of the basic floating-point operations and comparisons, rounding modes, floating-point exceptions and their handling, and conversion between different arithmetics.

Details of the two main floating-point precisions are shown in Table 1. Binary floating-point numbers are *normalized*, i.e., the exponent and mantissa are adjusted such that all bits in the binary representation of the value occur to the right of the radix point and the leading bit is a “1”. Since every floating-point number is stored in this way,[1] the leading “1” does not need to be explicitly stored, effectively giving the mantissa one extra bit of precision for no cost.

Table 1: IEEE Single and Double precision.

Type	Size	Sign	Exponent	Mantissa
Single	32 bits	1 bit	8 bits	23 bits
Double	64 bits	1 bit	11 bits	52 bits

In single precision, the maximum relative representation error is 2^{-24} with a range of $\pm 2^{-126}$ to $\pm(2-2^{-23})\times 2^{127}$. While in double precision, the maximum relative representation error is 2^{-54} and the effective range is $\pm 2^{-1022}$ to $\pm(2-2^{-52})\times 2^{1023}$.

The standard specifies that all arithmetic operations are to be performed as if they were calculated to infinite precision and then rounded according to one of four modes. The default rounding mode is to round towards the nearest representable number. In the case of a tie where the result is exactly half way between two adjacent floating-point numbers, the least significant bit is chosen to be zero (rounding to even). The other modes are rounding towards plus or minus infinity (facilitates interval arithmetic) and rounding towards zero (truncation).

The result of an operation can be so large or so small that it falls outside the available floating-point range. This is called *range violation*. When the magnitude is too large, the result is said to *overflow* the available range; when it is too small, an *underflow* occurs. It is also possible that invalid operations may be attempted. The IEEE Standard handles these special situations by including quantities such as NaN (Not a Number) and infinity. Without special quantities like these, a good way to handle exceptional situations like division by zero[2] is not available other than to abort computation.

The IEEE Standard provides a flag for each kind of floating-point exception. This flag is raised each time its exception is signaled and stays raised until the program resets it. Programs may also test, save, and restore the exception flags.

[1]The IEEE Standard also uses *denormal* or *subnormal* numbers to permit gradual underflow instead of flushing all underflows to zero.

[2]It is desirable to continue execution after a divide by zero when calculating trigonometric functions. For example, computing $\exp(\exp(-1.0/\cos(\chi)))$ when χ is a multiple of $\pi/2$ benefits from continuing execution. This arises when computing Chapman functions in the theory of planetary atmospheres [10].

2.1 SGI Origin 2000 Floating-Point Representation

On the O2K, single and double precision floating-point numbers are represented as specified by the IEEE standard. Thus, the single precision floating-point format allows representation to about seven significant decimal digits and double precision to about 15.

The O2K also supports long double precision, what SGI calls quad precision. This representation is not IEEE compliant. The quad precision has a 107-bit mantissa with an 11-bit exponent and one sign bit. In terms of decimal precision, this is approximately 34 decimal digits with a range of 10^{-292} to 10^{308}. (See [4] or the `math` man page on the O2K for more information.)

The rounding modes can be obtained and set via intrinsic functions. The function `get_ieee_rounding_mode()` returns the current floating-point rounding mode, and the function `set_ieee_rounding_mode()` alters the current floating-point rounding mode state and can be used to restore the floating-point rounding mode before exiting a procedure.

The example routine in Figure 1 calculates upper and lower bounds on a sum of numbers. This subroutine uses intrinsic subroutines to store, alter, and restore the floating-point

```
      subroutine sumupdn( n, x, xup, xdn )
         use ftn_ieee_definitions
         real x(n), xup, xdn
         integer save_rounding_mode
!        Save the current rounding mode then change the rounding mode
!        to round to positive infinity.
         call get_ieee_rounding_mode(save_rounding_mode)
         call set_ieee_rounding_mode(ieee_rm_pos_infinity)
!        Calculate the upper bound on sum.
         xup = sum( x )
!        Reset to round to negative infinity mode.
         call set_ieee_rounding_mode(ieee_rm_neg_infinity)
!        Calculate lower bound on sum.
         xdn = sum( x )
!        Restore original rounding mode and return to the caller.
         call set_ieee_rounding_mode(save_rounding_mode)
         return
      end subroutine sumupdn
```

Figure 1: Subroutine that calculates upper and lower bounds on a sum of numbers.

rounding mode. The current rounding mode is first stored in `save_rounding_mode`, and then the rounding mode is set to round towards infinity. This will round up all operations, thus producing an upper bound on the sum of the numbers in the array `x`. The rounding mode is then similarly altered to round towards negative infinity, thus giving a lower bound on the sum. The rounding mode is reset using the value stored in `save_rounding_mode`.

3 Detecting Exceptions on the SGI Origin 2000

When an exception occurs, the system responds in one of two ways:

- If the exception's trap (or interrupt) is disabled (default), the system records that an exception occurred and continues execution using the default specified by the IEEE Standard for the excepting operation.

- If the exception's trap is enabled, the system generates a `SIGFPE` signal. The trap handler `handle_sigfpes()` and the environment variable `TRAP_FPE` are provided in the library `libfpe` to unmask and handle these conditions.

The IEEE Standard defines five basic types of floating-point exceptions:

- Inexact - exact value cannot be represented in chosen precision.
- Underflow - result falls below range of precision.
- Overflow - result falls above range of precision.
- Divide by zero - result is ±infinity.
- Invalid operation - result is not a valid number.

The last three exceptions usually should not be ignored when they occur. The first two exceptions, inexact and underflow, commonly occur and can often be ignored. Note that most floating-point operations incur the inexact exception. Table 2 lists these five exceptions and their default result. The default values depend on the rounding mode. For instance,

Table 2: IEEE floating-point exceptions.

Exception Type	Examples	Default Value
Inexact	2.0/3.0, log(1.1)	Rounded result
Underflow	exp(−92.0)	Subnormal numbers or zero
Overflow	exp(92.0)	$\pm$`huge(1.0)`, $\pm\infty$
Divide by zero	$x/0$ for $x \neq 0, \pm\infty$, NaN	$\pm\infty$
Invalid operation	$0/0$, $0 \times \infty$	NaN (Not a Number)

assume the rounding mode is round to zero or round to negative infinity. With the intrinsic function `huge`,[3] the sum

$$\texttt{huge}(1.0) + \texttt{huge}(1.0)$$

will result in `huge`(1.0); while if the rounding mode is round to nearest or round to infinity, then the result is infinity.

By default, exceptional operations are masked. That is, the default value in Table 2 is returned as the result of the operation and the program continues silently. However, this event may be intercepted by causing an exception to be raised; i.e., an interrupt. When this occurs, the operating system generates a `SIGFPE` signal.

As required by the IEEE standard, the floating-point environment on the O2K provides users with a way to read and write the status flags. The flags are "sticky" in that once set,

[3]Fortran 90 intrinsic function that returns the largest number in the real numeric model.

they remain set until cleared. Note that the flags provide a way to differentiate an overflow from a genuine infinity.

The status flags can be tested to detect which exceptions have occurred and can also be explicitly set and cleared. The `get_ieee_exceptions()` function provides one way to access these flags, while `set_ieee_exceptions()` may be used to set the exceptions flags. Note that `get_ieee_status()` and `set_ieee_status()` may also be used to get and set the exceptions.

The routine `get_ieee_exceptions()` returns an integer value that combines all of the exception flags that have been raised. This value is the bitwise "OR" of the accrued exception flags where each flag is represented by a single bit. The bit position for each exception is shown in Table 3. Table 3 also shows the bit positions for the corresponding interrupts.

Table 3: Exceptions and Interrupts with their corresponding bit position.

Exception	bit
Invalid operation	16
Divide by zero	15
Overflow	14
Underflow	13
Inexact	12

Interrupt	bit
Invalid operation	11
Divide by zero	10
Overflow	9
Underflow	8
Inexact	7

Interrupts can be used to terminate execution of a program when floating-point exceptions occur (see Section 4).

The example in Figure 2 decodes the integer returned by `get_ieee_exceptions()`. It uses the Fortran 90 intrinsic function `btest()` that tests a specified bit of an integer value. A value of `.TRUE.` is returned if the bit is "1", `.FALSE.` if the bit is "0". This code fragment will print out a true or false value for each exception flag.

```
      integer status
      logical invalid, division, over, under, inexact
      call get_ieee_exceptions(status)
!   or call get_ieee_status(status) to get exceptions and interrupts
      invalid = btest(status,16)
      division = btest(status,15)
      over = btest(status,14)
      under = btest(status,13)
      inexact = btest(status,12)
      print*,invalid,division,over,under,inexact
```

Figure 2: Decoding the IEEE exception status flag.

Alternatively, the function `test_ieee_exception()` can be used to decode the value of `status`. This function is one of the intrinsic subroutines that support IEEE floating-point arithmetic on the O2K. It also returns a logical result. The following code fragment demonstrates use of this function.

```
      if( test_ieee_exception(ieee_xptn_inexact_result) ) &
          print*,'Inexact result'
```

The argument of `test_ieee_exception()` can be one of the five named constants

- `ieee_xptn_inexact_result`,
- `ieee_xptn_underflow`,
- `ieee_xptn_overflow`,
- `ieee_xptn_div_by_zero`,
- `ieee_xptn_invalid_opr`

provided in the `ftn_ieee_definitions` module.

4 Locating Exceptions on the SGI Origin 2000

When an exception is detected, most programmers want to know where the exception occurred. One way to accomplish this is to test the exception flags at various points in a program using the functions presented in the previous sections to isolate the exception precisely. This method is tedious with much overhead.

It is much simpler to determine where an exception occurs by enabling its interrupt. When an exception whose interrupt is enabled occurs, the program is sent a `SIGFPE` signal by the IRIX operating system. With the interrupt enabled, one can determine where the exception occurs by running under a debugger and stopping execution upon receipt of the `SIGFPE` signal or by using a `SIGFPE` handler that prints information about the instruction where the exception occurred.

The following subsections show how to use dbx, TotalView, and a floating-point exception handling library on the O2K to locate floating-point exceptions.

4.1 Using dbx

In order to use dbx (source-level debugger) to locate the instruction where a floating-point exception occurred, first insert the line

```
      call set_ieee_interrupts(ieee_ntpt_type)
```

where *type* is one of `invalid_opr`, `div_by_zero`, `overflow`, `underflow`, or `inexact_result`, near the beginning of the code (must be before the exception.) Then, as usual when using dbx, compile the code with the `-g` option. Finally, start up dbx (i.e., `dbx a.out`) and type `run`. The output will show the line where the exception occurred.

The program in Figure 3 can be used to illustrate this procedure. Compile this program with `f90 -g ex1.f`. This program runs to completion with the output

```
> a.out
 -6.,  NaN
```

```
      program ex1
      double precision x, y, sqrtm4
      x = -6.0d0
      y = sqrtm4(x)
      print * , x, y
      end

      double precision function sqrtm4(x)
      double precision x
      sqrtm4 = sqrt(x) - 4.0d0
      return
      end
```

Figure 3: Sample program with invalid exception.

Somewhere, an invalid operation exception occurred. Now, insert the line

```
      use ftn_ieee_definitions
```

to include named constants for use with the exception and interrupt intrinsic functions and insert the subroutine call

```
          call set_ieee_interrupts(ieee_ntpt_invalid_opr)
```

after the declarations, but before the assignment to `y`. Figure 4 shows the modified version of program `ex1`. Recompile the new version as before. Running this program now produces the output:

```
> a.out
Floating Exception
Abort (core dumped)
```

To find the cause of the invalid operation, use dbx to locate where the `SIGFPE` signal originates. The dbx session looks like the following.

```
> dbx a.out
dbx version 7.2.1.3m Dec 23 1998 01:09:37
Core from signal SIGABRT: Abort (see abort(3c))
(dbx) run
Process 834693 (a.out) started
Process 834693 (a.out) stopped on signal SIGFPE: Floating point exception (h
andler sigfdie) at [sqrtm4:12 +0x8,0x1000135c]
  12  sqrtm4 = sqrt(x) - 4.0d0
(dbx) print x
-6.0
(dbx)
```

```
program ex1
use ftn_ieee_definitions
double precision x, y, sqrtm4
call set_ieee_interrupts(ieee_ntpt_invalid_opr)
x = -6.0d0
y = sqrtm4(x)
print * , x, y
end

double precision function sqrtm4(x)
double precision x
sqrtm4 = sqrt(x)
return
end
```

Figure 4: Sample program with invalid operation interrupt enabled.

This shows that the exception occurred in the routine `sqrtm4` as a result of attempting to compute the square root of -6.0.

4.2 Using TotalView

TotalView, an interactive source-level debugger with a graphical interface (but no command-line interface), can be used like dbx. Compile the code as above, but instead of running `dbx a.out`, use `totalview a.out`. Execute the program from within TotalView. TotalView will stop at the exception showing the line and value of the offending variable. The user can set *breakpoints* in the code if the cause of the exception must be traced. As with dbx, this requires enabling interrupts to ensure that execution halts at the exception; otherwise the program will run to completion.

4.3 Using a Signal Handler to Locate an Exception

The previous subsections demonstrated that if trapping is enabled without a `SIGFPE` handler, the program aborts on the next occurrence of the trapped exception. Alternatively, if a `SIGFPE` handler is installed, the next occurrence of the trapped exception will cause the system to transfer control to a handler routine, which can print diagnostic information and either abort or resume execution.

On the O2K, a user can provide his own trap handler or can use the trap handler in the floating-point exception library `libfpe`. The library `libfpe` provides two methods to unmask and handle these conditions: the subroutine `handle_sigfpes()` and the environment variable `TRAP_FPE`. Both methods provide mechanisms for unmasking each condition (except inexact), for handling and classifying exceptions, and for returning either a default value or a chosen value for the offending operation. Mechanisms to count, trace, exit, or abort on enabled exceptions are also provided.

Figure 5 shows how to call `handle_sigfpes()`.

```
      program ex2
      use ftn_ieee_definitions
#include <f90sigfpe.h>
      double precision x, y, z, sqrtm4
      call my_sigfpes()
      x = -6.0d0
      y = sqrtm4(x)
      z = 1.0/(x+6.0)
      print * , x, y, z
      end

      double precision function sqrtm4(x)
      double precision x
      sqrtm4 = sqrt(x)
      return
      end

      subroutine my_sigfpes()
      use ftn_ieee_definitions
#include <f90sigfpe.h>
!     sets all underflows to zero
      fsigfpe(FPE_UNDERFL)%repls = FPE_ZERO

!     stack trace of first 5 exceptions of division by zero or invalid
      fsigfpe(FPE_DIVZERO)%trace=5
      fsigfpe(FPE_INVALID)%trace=5

!     counts each trap, printed every 100th trap
      fsigfpe(FPE_DIVZERO)%count=100
      fsigfpe(FPE_INVALID)%count=100

!     core dump and abort on second divide by zero or invalid
      fsigfpe(FPE_DIVZERO)%abort=5
      fsigfpe(FPE_INVALID)%abort=5

      call handle_sigfpes(FPE_ON, FPE_EN_UNDERFL + &
                          FPE_EN_DIVZERO +  FPE_EN_INVALID, &
                          0, FPE_ABORT_ON_ERROR, 0)
      return
      end
```

Figure 5: Sample code that calls the exception handler.

Compile the code with `f90 ex2.F -lfpe` and execute it. The following output is produced.

```
 -6.,  NaN,  Infinity

******* TRAP STATS FOR PID 840366 *********
     DIVZERO   1
     INVALID   1

   bad sig count = 0
   bad code count = 0
******* END TRAP STATS FOR PID 840366 *****
```

For sake of brevity, trace information (dbx stack trace) for each exception has been left out. The trace shows where the exceptions occurred. Also note that using `handle_sigfpes()` supersedes any interrupt calls located in the code. The environment variable `TRAP_FPE` should be undefined if a call to `handle_sigfpes()` is used.

4.4 Using the `TRAP_FPE` Environment Variable

If the code has been linked with the `libfpe` library, as above, then the runtime startup routine will check for the environment variable `TRAP_FPE`. The string read as the value of `TRAP_FPE` will be interpreted, and `handle_sigfpes()` will be called with the resulting values. Now, setting `TRAP_FPE` will not supersede hard-coded settings. So, for the example above, if the environment variable is set, the settings in `my_sigfpes()` will supersede those in the environment variable. However, the traps enabled in the previous example can be accomplished much easier with the environment variable.

For example, remove the call to `my_sigfpes()` in the previous program (Figure 5). Set the `TRAP_FPE` environment variable with the following options; for C shell or T shell,

```
   setenv TRAP_FPE "UNDERFL=ZERO; DIVZERO=COUNT(100),TRACE(5),ABORT(5);
                    INVALID=COUNT,TRACE(5),ABORT(5)"
```

or, for Korn shell,

```
   export TRAP_FPE="UNDERFL=ZERO; DIVZERO=COUNT(100),TRACE(5),ABORT(5);
                    INVALID=COUNT,TRACE(5),ABORT(5)"
```

Upon execution, the same exceptions are traced and counted as in Section 4.3. Setting the `TRAP_FPE` environment variable supersedes the use of calls to `set_ieee_interrupts()`.

4.5 Using SpeedShop

Another alternative is to use SGI's SpeedShop Toolbox. This multipurpose tool can be used to locate where floating-point exceptions occur. If the program is linked to the `libfpe` library, then the exception-handling facilities of the SpeedShop tools can be invoked either through a command line or window interface.

After compilation of the program, run SpeedShop on the executable with the the command `ssrun -fpe a.out`. A summary of all floating-point exceptions specified by the `TRAP_ENV` variable will be part of the output generated.

The command `cvd a.out` starts the window interface of SpeedShop. Once the workshop debugger window appears, choose "select task: find floating point exceptions" in the perf option and run the program. The locations where any floating point exceptions occurred will be identified.

5 Concluding Remarks

Floating-point exceptions can easily be trapped to show where the exception occurs on the SGI Origin 2000. This can be done using dbx, TotalView, or through exception handlers.

Calling the subroutine `handle_sigfpes()` is a preferred method when preparing software for others to use on the O2K, since it relieves the user of any need to know about the `TRAP_FPE` environment variable. The environment variable is preferable if one wants to experiment or allow any user to experiment with different trap behaviors with minimum effort or allow the user to have portable code.

A thorough reference on floating-point arithmetic is *What Every Computer Scientist Should Know About Floating-Point Arithmetic* by Goldberg [5]. Sun Microsystems' *Numerical Computation Guide* [9] is a valuable reference on IEEE arithmetic and exception handling. The *Fortran Language Reference Manual, Volume 2* [3] contains a detailed list of named constants and intrinsic exception functions that support IEEE floating-point arithmetic on the O2K. For related discussions on how to obtain faster numerical algorithms via exception handling, see [1, 6].

Acknowledgments

This work was supported in part by a grant of computer time from the DoD High Performance Computing Modernization Program at the ERDC MSRC, Vicksburg, MS.

References

[1] Demmel, J.W. and Li, X., Faster Numerical Algorithms via Exception Handling, *IEEE Trans. Comput.*, vol. 43, 1994, pp. 983-992. LAPACK Working Note #59.

[2] *Fortran Language Reference Manual, Volume 1.* Silicon Graphics, Document Number 007-3692-002 (`http://techpubs.sgi.com/library`).

[3] *Fortran Language Reference Manual, Volume 2.* Silicon Graphics, Document Number 007-3693-002 (`http://techpubs.sgi.com/library`).

[4] *Fortran Language Reference Manual, Volume 3.* Silicon Graphics, Document Number 007-3694-002 (`http://techpubs.sgi.com/library`).

[5] Goldberg, D., What Every Computer Scientist Should Know About Floating-Point Arithmetic, *ACM Comput. Surveys*, vol. 23, 1991, pp. 5-48.

[6] Hull, T.E., Fairgrieve, T.F., and Tang, P.T.P., Implementing Complex-Elementary Functions Using Exception Handling, *ACM Trans. Math. Soft.*, vol. 20, 1994, pp. 215-244.

[7] *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985.* Institute of Electrical and Electronics Engineers, New York, 1985. Reprinted in SIGPLAN Notices, 22(2):9-25, 1987.

[8] Kahan, W., *Lecture Notes on the Status of IEEE 754*, May 31, 1996.

[9] *Numerical Computation Guide*, Sun Microsystems, `http://www.sns.ias.edu/Main/computing/compilers_html/common-tools/numerical_comp_guide/index.html`.

[10] Schulze, D., Division by zero is OK in many cases, *NA Digest*, October 12, 1993, vol. 93: issue 38. `http://www.netlib.org/na-digest-html/93/v93n38.html`